Haaga-Helia
ammattikorkeakoulu Oy

JavaScript ja yksikkötestaus

Teemu Kälviäinen

Opinnäytetyö
Tietojenkäsittely
2016

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

14.2.2016

Tekijä(t)
Teemu Kälviäinen

Koulutusohjelma
Tietojenkäsittely

Opinnäytetyön otsikko
JavaScript ja yksikkötestaus

Sivu- ja liitesivumäärä
22 + 2

Opinnäytetyön otsikko englanniksi
JavaScript and unit testing

Opinnäytetyön tarkoituksena oli selvittää, miten JavaScript-koodia voidaan yksikkötestata ja että kuinka mielekästä JavaScriptin yksikkötestaus ylipäänsä on.

Opinnäytetyössä toteutettiin yksinkertainen prototyyppisovellus sekä laadittiin sille asianmukaiset yksikkötestit. Prototyyppisovelluksen yksikkötestausprosessi dokumentoitiin ja tämän pohjalta analysoitiin JavaScriptin yksikkötestauksen mielekkyyttä. Opinnäytetyön teoriaosuudessa on selvitetty lyhesti teoriatausta JavaScript-ohjelmointikielestä, yksikkötestauksesta sekä testien suunnittelusta.

Prototyyppisovelluksen testausprosessi osoitti JavaScriptin yksikkötestauksen olevan haastavaa, mutta pääosin mahdollista. Valitut yksikkötestaustyökalut - Jasmine ja QUnit - todettiin toimiviksi. Jatkotutkimusmahdollisuutena voisi olla syvällisempi tutustuminen JavaScriptin yksikkötestaukseen, esimerkiksi testiautomaation tai jatkuvan integraation näkökulmasta.

Asiasanat
Yksikkötestaus, JavaScript, ohjelmointi

Haaga-Helia
University of Applied Sciences

Abstract

14.2.2016

Author(s)
Teemu Kälviäinen

Degree programme
Business Information Technology

Report/thesis title	**Number of pages and appendix pages**
JavaScript and unit testing	22 + 2

The goal of this thesis was to resolve how JavaScript code can be unit tested and how meaningful it is to unit test JavaScript code in general.

In this thesis, a project application was developed and appropriate unit tests were constructed for it. Unit testing process was documented and the meaningfulness of JavaScript unit testing was analysed based on this. The theoretical background about JavaScript programming language, unit testing and test planning is shortly described in theoretical part of this thesis.

The testing process showed that unit testing JavaScript is challenging but, for the most part, possible. Selected tools, Jasmine and QUnit, were found functional. The next phase of the study could be a more comprehensive research into JavaScript unit testing, for instance from the test automation or continuous integration point of view.

Keywords
Unit testing, JavaScript, programming

Sisällys

1 Johdanto

Opinnäytetyössä toteutettiin prototyyppisovellus JavaScript-ympäristössä, sekä laadittiin sille asiankuuluvat yksikkötestit. Tarkoituksena oli selvittää miten JavaScript-koodin yksikkötestaaminen tapahtuu ja kuinka mielekästä JavaScriptin yksikkötestaaminen ylipäänsä on.

JavaScript on suosittu web-ohjelmointikieli, jota käytetään nykyään jo valtaosassa web-sovelluksia. JavaScriptin avulla verkkosivuille voidaan lisätä interaktiivisuutta ja dynaamista toiminnallisuutta.

Testausta pidetään tärkeänä osana ohjelmistotuotantoprosessia, ja eräs käytetyimmistä testausmenetelmistä on yksikkötestaus. Yksikkötestauksessa ohjelmaa testataan pieni osa kerrallaan, ja testeistä pyritään tekemään muista osista riippumattomia.

Opinnäytetyön tekijä on kiinnostunut erityisesti web-ohjelmoinnista, ja oli kiinnostunut kokeilemaan, kuinka hyvin yksikkötestaus onnistuu JavaScript-ympäristössä.

2 Teoria

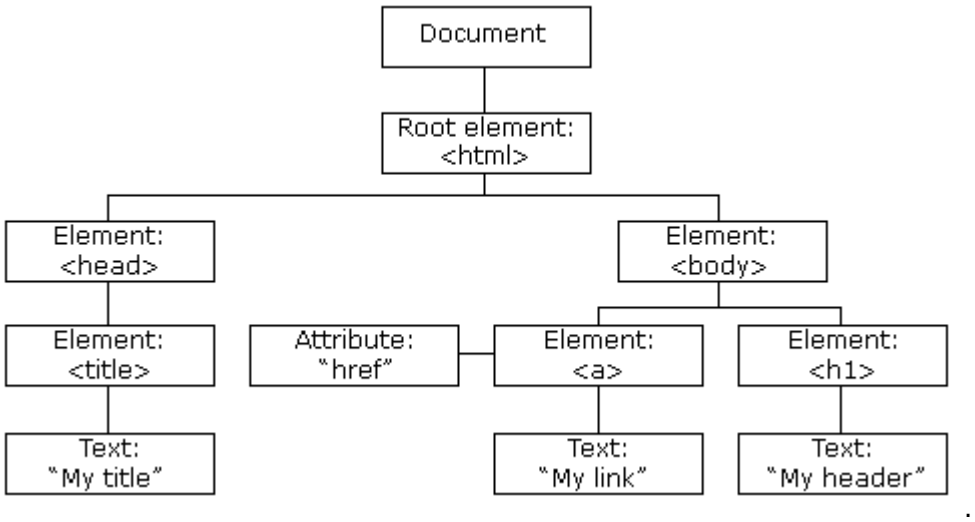
Tässä kappaleessa selvitetään teoriatausta sekä JavaScript-ohjelmointikielestä että yksikkötestauksesta. Teorian ymmärtäminen ennen tämänkaltaiseen työhön ryhtymistä on erittäin tärkeää työn onnistumisen kannalta.

2.1 JavaScript

JavaScript on web-ohjelmointikieli, joka on käytössä valtaosassa moderneja web-sovelluksia. Flanaganin (2011, 1) mukaan JavaScript on yksi kolmesta tekniikasta, joita kaikkien web-ohjelmoijien tulisi opetella: näistä HTML määrittää verkkosivuston sisällön, CSS tyylit ja JavaScript käyttäytymisen.

Teknisessä mielessä JavaScript on oliosuuntautunut ja heikosti tyypitetty. Oliosuuntautuneisuus näkyy esimerkiksi siinä, että lähes kaikki JavaScript-muuttujat ovat itse asiassa olioita; esimerkiksi merkkijonoilla on JavaScript-ohjelmointikielessä ominaisuus length, joka kertoo merkkijonon pituuden, sekä valmiiksi määriteltyjä metodeja kuten toUpperCase() ja substring(). Heikosti tyypitettyys näkyy puolestaan siinä, että JavaScript-ympäristössä muuttujia ja dataa voidaan helposti muuttaa tietotyypistä toiseen. Vahvasti tyypitetyissä ohjelmointikielissä muuttujille pitää tavallisesti määrittää tietotyyppi (esimerkiksi ”merkkijono” tai ”kokonaisluku”), mutta JavaScript-ympäristössä tällaista tarvetta ei ole (Ullman 2012, 4).

JavaScript-koodi pääsee käsittelemään HTML-dokumenttia DOMin eli Document Object Modelin avulla. DOM kuvaa verkkosivun rakennetta puumaisesti. DOMin avulla JavaScript pystyy luomaan dynaamista HTML:ää, esimerkiksi muokkaamaan ja poistamaan sivuston HTML-elementtejä sekä CSS-tyylejä (W3Schools).



Kuva 1. DOM-puu (W3Schools)

JavaScriptille on saatavilla useita erilaisia kirjastoja ja sovelluskehyksiä. Tämän opinnäytetyön kontekstissa itse JavaScript-koodi pyrittiin pitämään melko yksinkertaisena, mutta testauksessa käytettiin tarkoituksenmukaisia kirjastoja ja sovelluskehyksiä.

Nimien samankaltaisuudesta huolimatta JavaScript on täysin erilainen kuin Java-ohjelmointikieli (Flanagan 2011, 1).

2.2 Yksikkötestaus

Testaus on erittäin tärkeä osa ohjelmistokehitysprosessia. Testausta suoritetaan, sillä ohjelmistokehittäjät haluavat tietää ohjelmansa laadun, sekä jotta mahdolliset ongelmakohdat löytyisivät ajoissa. Huonosti suoritetulla testauksella voi olla tuhoisia seurauksia, sillä ohjelmaan on voinut jäädä virheitä, jotka olisi muutoin huomattu jo kehitysvaiheessa.

Ohjelmiston testaus voidaan jakaa useilla eri tavoilla erilaisiin vaiheisiin tai menetelmiin. Eräs tapa on jako yksikkötestaukseen, komponenttitestaukseen, integraatiotestaukseen ja järjestelmätestaukseen (Pan, 2016). Tässä opinnäytetyössä käsitellään yksikkötestausta.

Yksikkötestauksella tarkoitetaan testausmenetelmää, jossa käytetään niinsanottuja yksikkötestejä. Perinteisen määritelmän mukaan yksikkötesti on koodinpätkä, joka kutsuu toisaalla olevaa ohjelmakoodia ja tarkistaa toimiiko se ennaltaodotetulla tavalla (Saleh 2013, Chapter 1). Tällainen yksikkötesti voisi esimerkiksi tarkistaa, kutsutaanko ohjelman alustuksessa oikeaa funktiota. Yleensä yksikkötestauksen lisäksi käytetään myös muita testausmenetelmiä.

Yksikkötestauksen merkitys ohjelmistokehityksessä on suuri. Ilman yksikkötestejä ohjelmakoodin virheiden etsimisestä tulee todella työlästä, eikä virheenkorjauksen jälkeen voida varmistua, ettei sama virhe toistu ohjelmistokehityksen myöhäisemmässä vaiheessa (Saleh 2013, Chapter 1). Yksikkötestauksessa tulisi pyrkiä mahdollisimman suureen testikattavuuteen, mutta sadan prosentin testikattavuuteen on usein mahdotonta tai vaikea päästä.

Eräs yksikkötestaukseen läheisesti liittyvä termi on testivetoinen kehitys, josta käytetään usein englanninkielistä nimitystä ”test-driven development” tai lyhennettä TDD. Testivetoisessa kehityksessä testit kirjoitetaan ennen varsinaista ohjelmaa. Tässä opinnäytetyössä ei sovellettu testivetoista kehitystä, mutta tuloksissa voi olla viitteitä JavaScriptin soveltuvuudesta testivetoiseen kehitykseen.

2.3 Testien suunnittelu

Onnistuneen testauksen kannalta myös testien ja testauksen suunnittelu on tärkeää.

Tyypillisesti ohjelmistoja testatessa määritellään sarja testitapauksia, joita vasten testausta suoritetaan. Hyvä testitapaus koostuu yhdestä tunnistettavasta ohjelmiston käyttötapauksesta. Testitapaukseen kirjoitetaan testin esiehdot, testissä suoritettava toiminta sekä odotetut lopputulokset. Testin jälkeen on tarpeellista dokumentoida, milloin testi on suoritettu, kuka sen suoritti, ja millainen lopputulos testistä saatiin (Jorgensen 2013, 4).

Testitapausten suorittaminen vaatii sopivien esiehtojen asettamista, testitapauksen toiminnan suorittamista, lopputuloksen tarkkailua sekä lopputuloksen vertailua odotettuun lopputulokseen (Jorgensen 2013, 5). Testien suunnittelussa onkin olennaista ja tärkeää suunnitella testitapaukset sellaisiksi, että niiden testaaminen tällä tavoin on mielekästä. Yksikkötestauksen kannalta on myös tärkeää, että tällainen testi todellakin testaisi yhtä asiaa ohjelmakoodissa.

3 Prototyyppisovellus

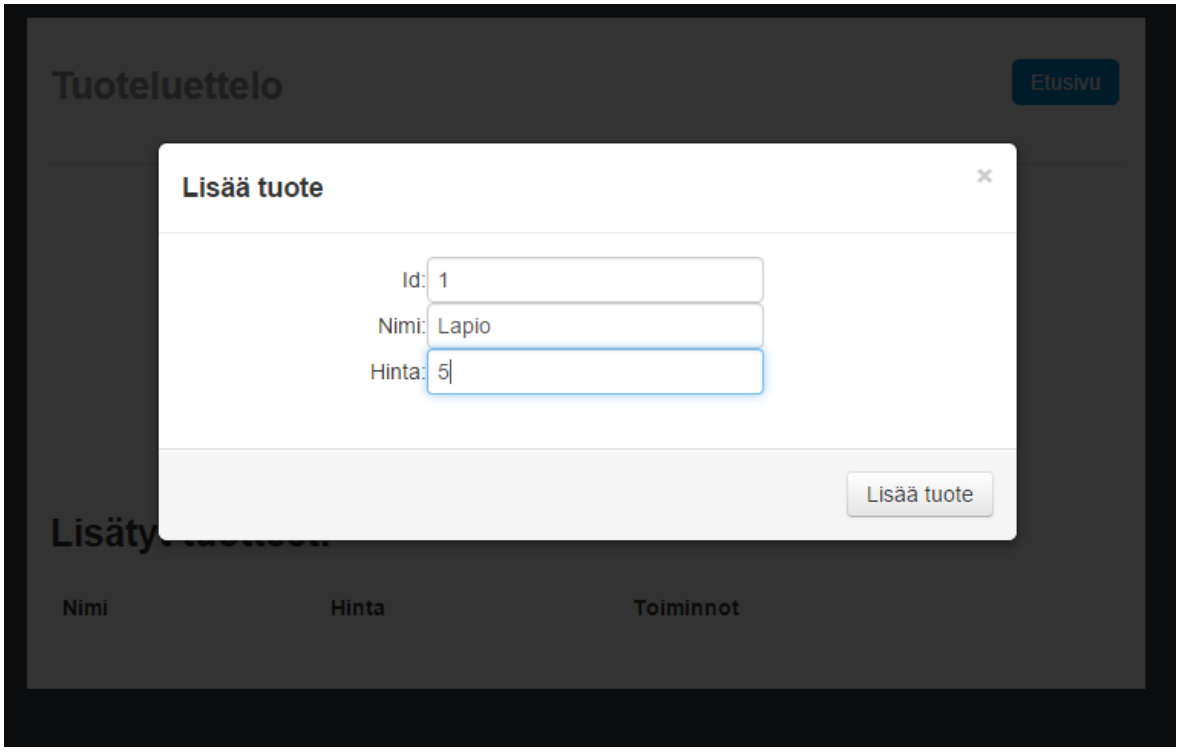
Tässä kappaleessa on selitetty lyhyesti opinnäytetyössä toteutetun prototyyppisovelluksen toteutuksesta, sekä siinä tehdyistä teknisistä ratkaisuista.

3.1 Sovelluksen toiminta

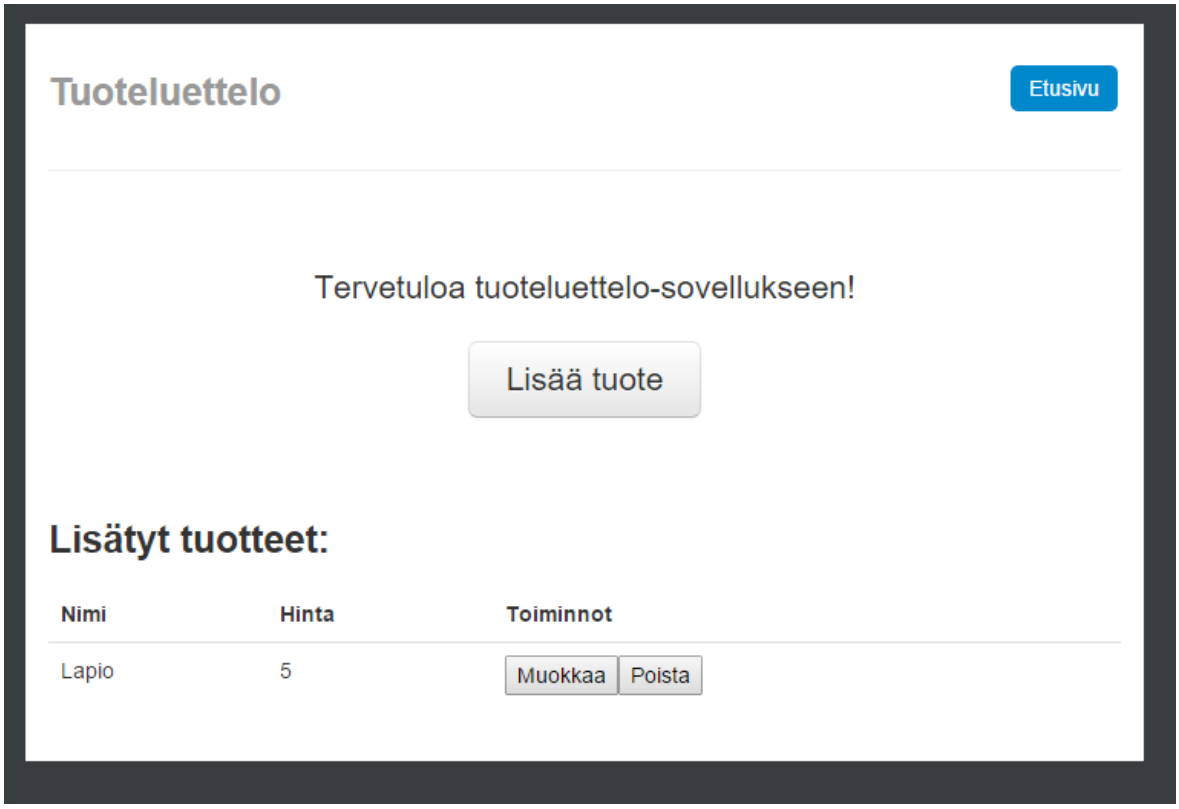
Prototyyppisovellus on yksinkertainen tuoteluettelo, joka tarjoaa mahdollisuuden lisätä tuotteita listalle, muokata tuotteita ja poistaa tuotteita. Sovellus on toteutettu ns. yhden sivun sovelluksena, joten eri toimintojen välillä ei liikuta pääsivulta muualle. Tuotteen lisäys- ja muokkausikkunat avautuvat niinsanottuun modal-ikkunaan. Modal on Bootstrap.js -kirjaston käyttämä nimitys eräänlaisesta ponnahdusikkunasta (Bootstrap, 2016). Tuotteiden luettelo on tulostettu sivun alalaitaan, jossa se on koko ajan näkyvissä.

Sovelluksessa ei ole toteutettu minkäänlaista tietokantayhteyttä, joten lisätyt, poistetut ja muokatut tuotteet eivät säily mikäli sivua päivittää. Tietokantayhteyden luominen ei ollut tämän opinnäytetyön kontekstissa tarpeellista, sillä opinnäytetyössä käsitellään JavaScript-ohjelmointikieltä. Mahdollinen tietokantayhteys sen sijaan luotaisiin jollakin muulla tekniikalla.

Prototyyppisovelluksen toimintaa on esitelty kuvissa 2, 3 ja 4.



Kuva 2. Tuotteen lisäys luetteloon



Kuva 3. Luettelo tuotteista, jossa näkyvissä äskettäin lisätty tuote.

Kuva 4. Lisätyn tuotteen muokkaus

3.2 Sovelluksessa käytetyt tekniikat

Prototyyppisovellus on toteutettu JavaScriptillä ja HTML:llä. Ohjelmakoodi on pyritty pitämään tarkoituksellisesti mahdollisimman yksinkertaisena, mutta puhtaan JavaScriptin lisäksi siinä on käytetty jQuery- ja Bootstrap-kirjastoja. Prototyyppisovellus kehitettiin Sublime Text -tekstieditoria käyttäen, joskaan tällä ei ole mitään vaikutusta testaamisen kannalta.

jQuery-kirjaston perusidean voisi tiivistää lauseeseen ”kirjoita vähemmän, tee enemmän”. Sen tarkoituksena on siis helpottaa JavaScriptin käyttöä verkkosivuilla. jQuery mahdollistaa monien yleisten tehtävien suorittamisen kutsumalla siinä olevia metodeita. Näin saadaan suoritetuksi tehtäviä, joiden toteuttaminen itse olisi työlästä ja vaatisi useita rivejä ohjelmakoodia (W3Schools). Tässä prototyyppisovelluksessa jQueryä on hyödynnetty lisäys- muokkaus- ja poistolomakkeiden yhteydessä.

Myös Bootstrap-kirjaston tarkoituksena on helpottaa ohjelmoijan työtä. Siitä on erityisesti hyötyä sivuston ulkoasun toteutuksessa, ja sen avulla on mahdollista toteuttaa helposti responsiivisia verkkosivuja (W3Schools). Prototyyppisovelluksessa Bootstrapia on käytetty modal-ikkunan avaamiseen (katso kuvat 2 ja 4). Lisäksi sovelluksen CSS-tyylit ovat pitkälti suoraan peräisin Bootstrap-kirjastosta.

4 Prototyyppisovelluksen testien suunnittelu

Tässä kappaleessa on esitetty prototyyppisovelluksen testien suunnittelua.

4.1 Testien suunnittelu ja käyttötapaukset

Testauksen ideana on selvittää, toimiiko sovellus odotetulla tavalla. Valitun testausmenetelmän, yksikkötestauksen, ideana puolestaan on jakaa sovellus pieniin yksiköihin, joita testataan erikseen. Ihanteellisinta olisi, että yksikkötestissä – nimensä mukaisesti – testattaisiin vain yhtä asiaa ohjelmakoodissa.

Ennen varsinaisten testien kirjoittamista on hyvä suunnitella testitapaukset etukäteen. Testauksessa olisi hyvä päästä mahdollisimman suureen testikattavuuteen, mutta tiettyjä asioita voidaan rajata testien ulkopuolelle. Prototyyppisovelluksessa yksikkötestauksen ulkopuolelle jätettiin Bootstrap- ja jQuery-kirjastojen testaus, sillä molemmat ovat kolmansien osapuolten toteuttamia, ja niiden kehittäjien voidaan olettaa testanneen niiden toimivuuden.

Testien suunnittelun helpottamiseksi prototyyppisovellus jaettiin käyttötapauksiin, joiden avulla havainnollistettiin, mitä käyttäjä haluaisi sovelluksella tehdä. Seuraavissa kappaleissa on esitelty sovelluksen käyttötapaukset.

4.1.1 Käyttäjä haluaa katsella tuotteita

Käyttäjän tulee voida katsella tuoteluetteloa. Tuoteluettelon tulee tulla näkyviin sovellus avattaessa, ja pysyä näkyvillä koko ajan.

4.1.2 Käyttäjä haluaa lisätä tuotteen

Käyttäjän tulee voida lisätä tuote luetteloon avaamalla etusivulta tuotteen lisäyslomake. Lomakkeessa käyttäjä voi täyttää nimen ja hinnan, sovellus täyttää id-kentän automaattisesti. Kun käyttäjä painaa ”Lisää tuote” –painiketta, tuote ilmestyy luetteloon. Käyttäjän syöte tarkistetaan, ja sovellus ilmoittaa lisäyksen epäonnistumisesta, mikäli syötteessä on jotain vikaa.

4.1.3 Käyttäjä haluaa muokata tuotetta

Käyttäjän tulee voida muokata listalla olevaa tuotetta. Sovelluksen tulee hakea oikea tuote muokattavaksi lomakkeelle. Muokattavan tuotteen tulee saada lomakkeelta id, nimi ja

hinta. Muokatun tuotteen tiedot päivittyvät näytölle, kun käyttäjä painaa ”Tallenna” -painiketta.

4.1.4 Käyttäjä haluaa poistaa tuotteen

Käyttäjällä tulee olla mahdollisuus poistaa haluttu tuote listalta. Tuote poistuu luettelosta, kun käyttäjä painaa luettelon yhteydessä olevaa ”Poista” –painiketta.

4.2 Testauskirjastojen valitseminen

Prototyyppisovelluksen yksikkötestit päätettiin laatia Jasmine- ja QUnit-kirjastoilla. Mikäli kyseessä olisi todellinen käyttöön otettava sovellus, olisi sovellukselle valittu yksi testauskirjasto, jota olisi hyödynnetty koko testausprosessissa. Tämän opinnäytetyön tarkoituksena oli kuitenkin tutkia testaamista JavaScript-ympäristössä, joten testien kirjoittamistakin kokeiltiin eri kirjastoja hyödyntäen.

5 Prototyyppisovelluksen testaaminen

Tässä kappaleessa on kuvattu prototyyppisovelluksen testausprosessi. Lisäksi käytetyt testauskirjastot esitellään tarkemmin.

5.1 Tuotteen lisäämisen testaus Jasminella

Jasmine on JavaScript-koodin testaamiseen käytetty sovelluskehys. Jasmine ei vaadi toimiakseen mitään muita JavaScript-kirjastoja, eikä myöskään Document Object Modelia eli DOM:ia (Jasmine, 2015).

Jasmine-testien rakennetta voidaan kuvailla helpoiten esimerkin avulla. Prototyyppisovelluksen ensimmäisessä käyttötapauksessa määriteltiin, että sovelluksessa tulisi pystyä katselemaan listaa tuotteista. Toiminnallisuus on kuitenkin käytännössä toteutettu HTML:llä eikä JavaScriptillä, joten rajasin sen yksikkötestauksen ulkopuolelle ja aloitin testauksen käyttötapauksesta 2 eli tuotteen lisäämisestä. Tähän käyttötapaukseen liittyvä funktio on tässä tapauksessa add-funktio, jonka koodi on näkyvissä kuvassa 5.

```
// function that adds item to the list
ProductList.add = function(name, price, productid){
    newitem = null;

    var isPriceCorrect = ProductList.checkPrice(price);
    console.log("name" +name);
    console.log("price" +price);

    if ((isPriceCorrect) && name !== ""){
        newitem = {
            name:name,
            price:price,
            productid:productid};

        ProductList.items.push(newitem);
        $('.addModal').modal('hide');

        $('#errorarea').html(
            ""
        );

        $('#tuoteluettelo tr:last').after('<tr id="tr'+newitem.productid+'"><td>'+newitem.name+'</td><td>'+newitem.price+'</td><td>'+
            '<input type="button" onclick="ProductList.callGetAndOpenModal('+newitem.productid+')" value="Muokkaa" />'+
            '<input type="button" onclick="ProductList.removeitem('+newitem.productid+')" value="Poista" /></td></tr>');
    } else {
        $('#errorarea').html(
            "<p>There are errors in added product. Please fix errors and try again.</p>"
        );
    }
};
```

Kuva 5. add-funktio.

Funktiossa lomakkeen kenttien ”name” ja ”price” arvojen oikeellisuus tarkistetaan (nimen kohdalla tarkistetaan, ettei se ole tyhjä, pricen kohdalla taas suoritetaan tarkistus erillisessä funktiossa; productid:tä käyttäjä ei syötä itse, joten sitä ei tarkisteta tässä vaiheessa). Mikäli arvot ovat oikein, muodostetaan näistä arvoista olio, joka lisätään items-nimiseen taulukkoon. Mikäli arvoissa on jotakin pielessä, oliota ei muodosteta ja käyttäjälle annetaan virheilmoitus lomakkeen alapuolelle. Ensimmäisenä asiana

yksikkötestataan, että muodostaako sovellus tuotteen oikealla productid:llä. Jasminella laadittu yksikkötesti on esitetty kuvassa 6.

```
describe("Add", function() {

  it ("should add object with correct productid", function() {
    var productid = 1;
    ProductList.add("Nimi", 1, productid);
    expect(newitem.productid).toEqual(productid);
  });
```

Kuva 6. Jasminella laadittu yksikkötesti

Yksikkötestin describe-blokissa on kuvailtu, mitä testataan; tässä tapauksessa kyseessä on add-funktio. Yksikkötestissä oletetaan, että newitem-olio saa productid-arvoksi luvun 1. Testin suorittamalla voidaan tarkistaa, toimiiko sovellus tältä osin oletetulla tavalla.

Jasminen mukana tulee SpecRunner.html -niminen tiedosto, jonka suorittamalla voidaan tarkistaa testien tuloksia. Jasmine-testien ajamiseen olisi mahdollista käyttää myös muita menetelmiä, mutta tässä kontekstissa SpecRunner.html ajaa asiansa varsin hyvin. Testien suorittaminen vaatii ainoastaan, että kyseiseen tiedostoon lisätään viittaukset sekä testeihin, että testattavaan koodiin.

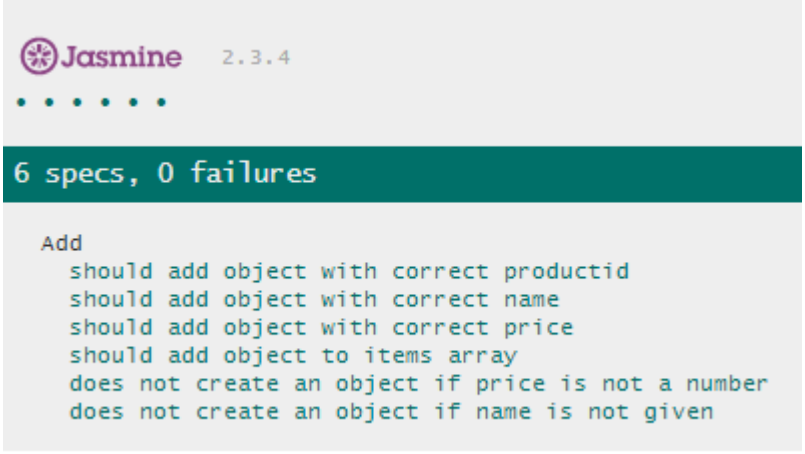
Tiedosto avaamalla nähty tulos on esitelty kuvassa 7.

Kuva 7. Onnistunut Jasmine-yksikkötesti

Yksikkötesti suoritettiin onnistuneesti. Tämän jälkeen oli luontevaa kirjoittaa yksikkötestit myös lopuista add-funktion toiminnoista. Testattaviin asioihin kuuluivat muut lomakkeen kentät sekä se, että tuote päätyy items-taulukkoon.

Yksikkötestit oli varsin helppo laatia niihin tapauksiin, joissa arvot ovat oletettuja. Yksikkötestauksen kannalta on kuitenkin tärkeää testata myös erilaisia virhetilanteita, eli esimerkiksi sitä, mitä tapahtuu jos käyttäjä syöttää virheellisiä arvoja tai jättää arvot

syöttämättä kokonaan. Laadinkin siis erikseen testit, joista toisessa ”price”-muuttujaan asetetaan tekstiarvo ja toisessa tuotteen nimi on tyhjä. Sain nämäkin testit menemään onnistuneesti läpi, ja Jasminen antama palaute kaikista onnistuneista testeistä on esitetty kuvassa 8.



Kuva 8. Onnistuneita Jasmine-yksikkötestejä

Testien ajamisen jälkeen havaittiin, että kaikki testit suoritettiin onnistuneesti. Myös manuaalinen testaus antoi samanlaisia tuloksia. Koska opinnäytetyön tarkoituksena on tutkia yksikkötestauksen mielekkyyttä JavaScript-ympäristössä, päätin lisätä tarkoituksella yksikkötestin, jonka tulisi epäonnistua. Muokkasin siis testiä, jonka tarkoituksena on testata tuotteen lisäämistä sellaiseksi, että testin olettama tulos poikkeaa oikeasta tuloksesta. Tällainen tahallisesti virheellinen testi on esitelty kuvassa 9.

```
it ("should add object with correct name", function() {
  var name = "Nimi";
  productList.add("Ihan väärä nimi", null, null);
  expect(newitem.name).toEqual(name);
});
```

Kuva 9. Esimerkki epäonnistuvasta Jasmine-testistä

Tässä yksikkötestissä oletetaan, että newitem-olio saa nimekseen muuttujan name arvon (”Nimi”), mutta add-funktiota kutsutaan nimellä ”Ihan väärä nimi”. Testejä suoritettaessa Jasminen tulisi ilmoittaa, että kyseinen testi epäonnistuu. Testien suorittamisen jälkeen auennut virheilmoitus on esitelty kuvassa 10.

Kuva 10. Epäonnistunut Jasmine-testi

Tuloksesta voidaan todeta, että Jasmine osaa suorittaa muut testit onnistuneesti, vaikka yksi menisikin pieleen – tiedämme siis, ettei koodissa muualla pitäisi olla mitään vikaa. Virheilmoituksesta voidaan myös havaita, että Jasmine oletti olion saavan nimekseen ”Nimi”, mutta saikin tuloksen ”Ihan väärä nimi”. Tässä tapauksessa virhe oli tarkoituksella aiheutettu ja itse yksikkötestissä, mutta normaalisti virhe olisi koodissa ja sitä lähdettäisiin etsimään sieltä. Esimerkin perusteella voidaan kuitenkin todeta, että Jasmine osaa ilmoittaa epäonnistuneista testeistä varsin hyvin.

Kokonaisuutena Jasmine vaikutti tämän lyhyen kokeilun perusteella oikein toimivalta yksikkötestauskirjastolta. Testien kirjoittaminen oli selkeää, ja Jasminen oma test runner osasi antaa riittävän hyvää palautetta sekä onnistuneista että epäonnistuneista yksikkötesteistä. Ennen lopullisia päätelmiä päätin kuitenkin kokeilla myös QUnit-testauskirjastoa.

5.2 Tuotteen muokkaamisen testaus QUnitilla

Toisena testaustyökaluna opinnäytetyössä tutkitaan QUnitia. QUnitin avulla voidaan – Jasminen tavoin – testata miltei mitä tahansa JavaScript-koodia. Huomionarvoista QUnitissa on muunmuassa se, että jQuery käyttää QUnitia yksikkötestaukseen. QUnitin alkuperäinen kehittäjä, John Resig, kehittikin sitä alunperin jQueryn osana. Nykypäivänä QUnit ei kuitenkaan enää vaadi jQueryä toimiakseen eikä se ole muutenkaan riippuvainen jQuerystä (QUnit, 2015).

Prototyyppisovelluksen käyttötapaus 2, tuotteen lisääminen, yksikkötestattiin Jasminella. QUnitilla testattavaksi asiaksi valikoitui siis käyttötapaus 3, tuotteen muokkaaminen.

Ensimmäiseksi QUnit-testiksi valikoitui, että sovellus hakee listalta oikean tuotteen muokattavaksi. Kyseisen toiminnallisuuden tekevä JavaScript-funktio hakee tuotteen items-taulukosta productid:n perusteella, muokkaa ”modal-edit-body”-id:llä olevaa elementtiä DOM:issa ja avaa modaali-ikkunan, jonka sisältönä on kyseinen elementti. Suunnitellessani funktion testaamista havaitsin, että funktiota voisi olla syytä muokata, sillä siinä suoritetaan mielestäni liian monta asiaa kerralla. Eriytin eri toiminnallisuuksia omiksi funktioikseen. Näiden muutosten jälkeen sovelluksen ylläpidettävyys parani, sillä uudistetut funktiot suorittavat vain yhden asian, ja esimerkiksi DOMia muokkaavaa funktiota voidaan nyt muokata ilman, että sillä on mitään vaikutusta tuotteen hakemiseen käytettyyn funktioon. Samalla tuli todistettua, että jo testien suunnittelu voi parantaa koodin laatua. En todennäköisesti olisi tullut huomanneeksi tätä asiaa, jos olisin vain laatinut prototyyppisovelluksen ilman yksikkötestejä.

getitem-funktio (johon oli tehty tarvittavat muutokset) on esitelty kuvassa 11.

```
ProductList.getitem = function (productid, items){
    if (items === undefined){
        items = ProductList.items;
    }

    var result = items.filter(function(obj) {
      return obj.productid == productid;
    });

    var itemtomodify = result[0];

    return itemtomodify;

};
```

Kuva 11. getitem-funktio

Korjausten jälkeen funktio ottaa parametreikseen productid-muuttujan sekä items-taulukon, käy läpi items-taulukon ja palauttaa sieltä ensimmäisen oikealla productid:llä löytämänsä olion. Muokatussa rakenteessa getitem ei ota kantaa items-taulukkoon muuten kuin siinä, että siellä olevissa olioissa on oltava productid-muuttuja.

QUnitissa ei – toisin kuin Jasminessa – tullut mukana testien tulokset näyttävää tiedostoa. Tällaisen luominen itse on kuitenkin suhteellisen helppoa. Käytännössä tiedostoon tarvitaan – samaan tapaan kuin Jasminen SpecRunner.html –tiedostossa – viittaukset QUnitiin, testattavaan koodiin, testeihin sekä QUnitin CSS-tyylitiedostoon.

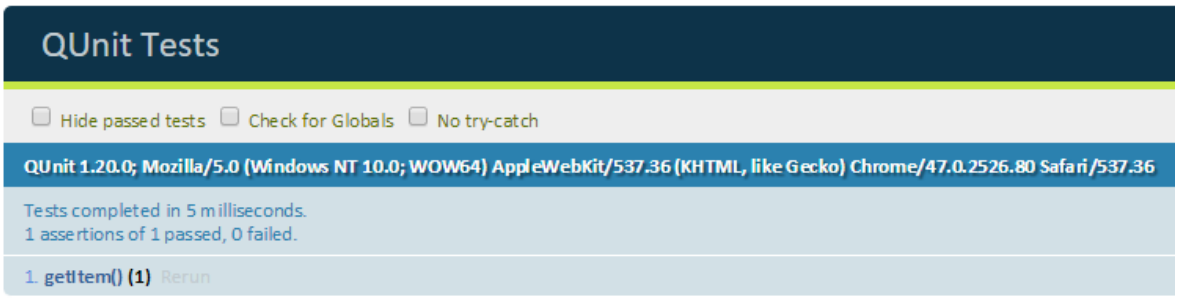
getitem-funktiota testaava QUnit-testi on esitelty kuvassa 12.

```
test('getitem()', function() {
    var items = [
    {productid:0, name:"First item", price:0},
    {productid:1, name:"Second item", price:1},
    {productid:2, name:"Third item", price:2}];

    var expecteditem = {productid:1, name:"Second item", price:1};

    var itemtomodify = ProductList.getitem(1, items);

    deepEqual(itemtomodify, expecteditem, 'Did not return correct item based on id');
});
```

Kuva 12. QUnit-testi

Testissä asetetaan testiä varten items-taulukkoon kolme oliota, sekä kutsutaan getitems-funktiota antamalla parametreiksi id ”1” ja items-luettelo. Testissä tarkistetaan, palauttaako getitem-funktio oletetun tuloksen, joka on määritelty expecteditem-oliossa.

Suorittamalla aiemmin laadittu QUnit-testien tulokset näyttävä HTML-tiedosto saadaan aikaan QUnit-testin onnistumisilmoitus, joka on esitelty kuvassa 13.



Kuva 13. Onnistunut QUnit-testi

Esimerkin vuoksi päätin jälleen kokeilla, millaista informaatiota QUnit antaa epäonnistuneista testeistä. Muutin ylläolevaa testiä yksinkertaisesti siten, että getitem-funktiota kutsutaankin productid:llä ”2”, jolloin testin tulisi epäonnistua, sillä oletetun tuloksen productid on ”1”. Virheilmoitus on esitelty kuvassa 14.

Kuva 14. Epäonnistunut QUnit-testi

Epäonnistuneen testin tuloksesta voidaan havaita, että QUnit osaa ilmoittaa hyvin tarkasti ja informatiivisesti, mikä testissä meni pieleen. Odotettu tulos näytetään vihreällä, ja ohjelman palauttama virheellinen tulos punaisella. Mikäli koodissa olisi oikeasti jotakin vikaa, QUnit paljastaisi sen erittäin selkeästi ja hyvin.

Kun QUnit oli täten todettu päteväksi yksikkötestaustyökaluksi, oli sillä luontevaa kirjoittaa myös muut käyttötapaukseen liittyvät testit. getitem-funktion jälkeen tuntui luontevalta testata callGetAndOpenModal –nimistä funktiota, jonka tarkoituksena on yksinkertaisesti kutsua edellä mainittua getItem-funktiota, sekä kutsua openmodal-funktiota antaen parametriksi getItem-funktion palauttaman olion.

Tätä testiä kirjoittaessani jouduin tutustumaan Sinon.js –kirjastoon, jonka avulla voidaan luoda spy-, stub- ja mock-olioita JavaScriptille. Spy tallentaa testissä tarvittavan funktion argumentit, palauttaman arvon sekä mahdolliset virheilmoitukset. Stub muistuttaa spyta, mutta sillä on esimääriteltyä toiminnallisuutta. Mock puolestaan muistuttaa kumpaakin, mutta mockilla on myös esimääriteltyjä odotuksia (Sinon.js, 2015).

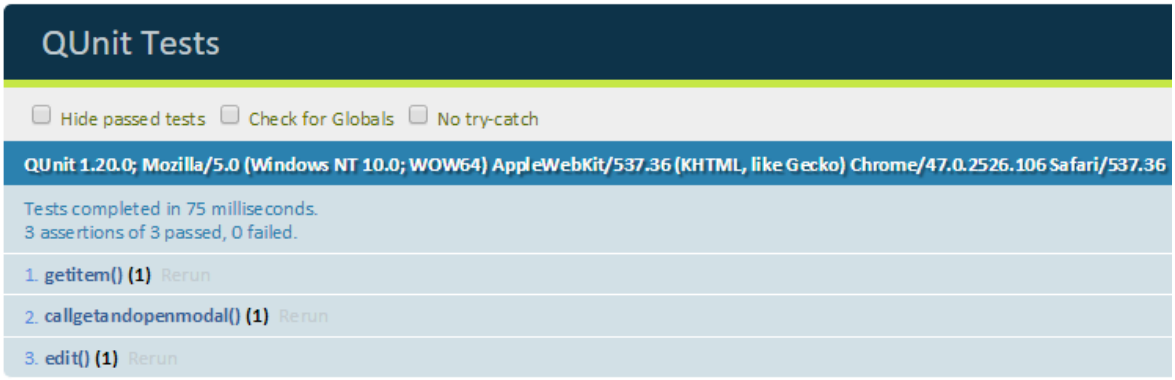
CallGetAndOpenModal-funktiota testatessani tarvitsin spyta tarkistaakseni, kutsutaanko openModal-funktiota getItem-funktiosta saadulla oliolla. Esimerkki spyn käytöstä on kuvassa 15.

```
sinon.spy(ProductList, "openModal");

ProductList.callGetAndOpenModal(1);

ok(ProductList.openModal.calledWith(item), "Did not call open modal with correct item");
```

Kuva 15. Spyn käyttöä yksikkötestissä

Varsinaisen muokkaustoiminnallisuuden eli edit-funktion testaaminen oli varsin yksinkertaista. Käytännössä kyseistä funktiota kutsutaan kuvassa 4 esitetystä lomakkeesta: funktio käy läpi items-taulukon, ja löydettyään täsmäävän id:n muokkaa tiedot lomakkeen kenttien mukaisiksi. Yksikkötesti muistuttaa huomattavan paljon getItem-funktion testiä: sillä erotuksella, että tässä tapauksessa funktiolle annetaan id:n lisäksi parametreiksi myös nimi ja hinta. Laadittuani testin se meni QUnitin mukaan läpi.

Katsoin näiden testien myötä muokkaustoiminnallisuuden testit valmiiksi. Testauskirjastona käytetty QUnit osoittautui Jasminen tavoin toimivaksi välineeksi yksikkötestaukseen. Sinon.js oli kiehtova lisä, jota en alun perin ajatellut sisällyttää opinnäytetyöhön, mutta joka osoittautui tarpeelliseksi testejä kirjoittaessa. QUnitin antama onnistumisilmoitus kaikista sille laadituista testeistä on esitelty kuvassa 16.



Kuva 16. Onnistuneita QUnit-testejä

Tässä vaiheessa testattavista asioista oli jäljellä vielä tuotteen poisto sekä id:n asettaminen tuotteelle. Päätin tehdä nämä testit Jasminella.

5.3 Tuotteen poiston ja id:n asettamisen testaaminen Jasminella

Tässä vaiheessa sovelluksesta oli testaamatta vielä kaksi funktiota: removeitem (joka hakee tuotteen productid-muuttujan perusteella, poistaa tämän items-taulukosta ja muokkaa DOM:ia poistaen taulukosta kyseisen tuotteen rivin) sekä setid, joka

yksinkertaisesti asettaa tuotteelle uuden id:n joka kerta, kun uutta tuotetta ollaan luomassa.

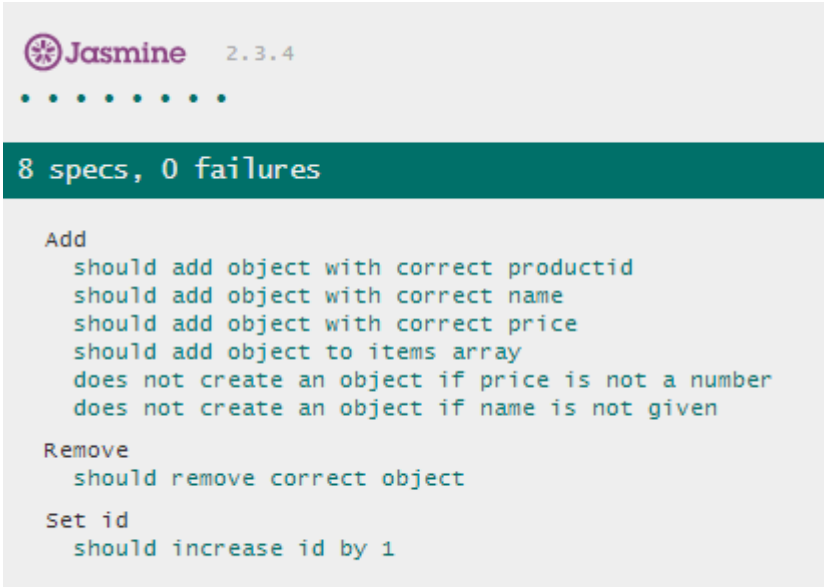
Removeitem-funktion testiä laatiessa huomasin tarvitsevani Jasminen beforeEach-funktiota, jota ei käytetty aikaisemmissa Jasmine-yksikkötesteissä. beforeEach-funktiossa tehdään alkuvalmisteluja, joita testeissä käytetään (Jasmine, 2015). Tässä tapauksessa käytin beforeEach-funktiota asettaakseni ProductList.items –taulukkoon valmiiksi olioita. Esimerkki beforeEach-funktiota käyttävästä testistä on kuvassa 17.

```
describe("Remove", function() {

   beforeEach(function() {
    ProductList.items = [
        {productid:0, name:"First item", price:0},
        {productid:1, name:"Second item", price:1},
        {productid:2, name:"Third item", price:2}];
  });

  it ("should remove correct object", function() {

    var expecteditems = [
        {productid:0, name:"First item", price:0},
        {productid:2, name:"Third item", price:2}];

    var productid = 1;
    ProductList.removeitem(productid);

    expect(ProductList.items).toEqual(expecteditems);
  });

});
```

Kuva 17. Testi, joka käyttää beforeEach-funktiota.

Testin mentyä läpi totesin, ettei funktiosta oikeastaan ole muuta testattavaa. Siirryinkin siis setid-funktion testaamiseen, joka osoittautui varsin yksinkertaiseksi, sillä funktion sisältönä on vain se, että sitä kutsuttaessa id-lukua kasvatetaan yhdellä ja asetetaan tuotteen lisäyslomakkeen Id-kenttään, joka on readonly-tyyppiä, eli käyttäjä ei itse voi tehdä siihen muutoksia. Käytin myös tässä testissä avukseni beforeEach-funktiota, jolla asetin id-muuttujalle alkuarvon.

Näiden testien laatimisen jälkeen Jasminen tulokset on esitelty kuvassa 18.



Kuva 18. Kaikki Jasmine-testit läpi

Totesin tässä vaiheessa prototyyppisovelluksen yksikkötestauksen valmiiksi. Prosessin onnistumista on arvioitu tarkemmin myöhemmissä luvuissa.

6 Muut JavaScript-testauskirjastot- ja menetelmät

Tämän opinnäytetyön puitteissa JavaScriptin yksikkötestausta kokeiltiin käytännössä Jasmine- ja QUnit-kirjastoilla sekä kokeiltiin lyhyesti Sinon.JS-kirjastoa. Menetelmät ja työkalut JavaScript-koodin yksikkötestaamiseen eivät kuitenkaan rajoitu näihin, ja tässä luvussa käydään lyhyesti läpi muita vaihtoehtoja.

Jasminen ja QUnitin lisäksi Mocha on mainitsemisen arvoinen JavaScript-testauskirjasto. Mocha eroaa edellä mainituista testauskirjastoista esimerkiksi siten, että se pyörii Node.js -alustalla (Mocha, 2015). Node.js on JavaScript-sovellusalusta, jota voidaan käyttää esimerkiksi palvelinpuolen JavaScript-sovelluksissa (Tutorialspoint, 2015). Node.js -alustasta johtuen Mochan käyttöönotto ei ole aivan yhtä helppoa ja yksinkertaista kuin Jasminen ja QUnitin, ja osittain tästä syystä tarkempi tutustuminen Mochaan ja Node.js:ään jäikin tämän opinnäytetyön ulkopuolelle.

Testauskirjastojen lisäksi JavaScriptin yksikkötestaamiseen liittyvät olennaisesti ns. test runnerit. Test runnerilla tarkoitetaan ohjelmaa, jota käytetään yksikkötestien suorittamiseen. Tämän opinnäytetyön esimerkeissä käytettiin Jasminen ja QUnitin omia työkaluja testien ajamiseen, mutta varsinkin laajemmissa sovelluksissa voi olla tarpeen käyttää erillisia test runnereita. Näistä mainittakoon tässä Karma.

Karma on AngularJS:n – erään suositun JavaScript-sovelluskehyksen - suosittelema työkalu yksikkötestien ajamiseen. Karma on Node.js-komentorivityökalu, joka suorittaa testejä palvelinympäristössä. Karman avulla yksikkötestejä voidaan esimerkiksi ajaa useita eri selaimia vasten, mikä voi olla hyödyllistä, sillä JavaScript-koodin toimivuudessa on eroa eri selaimilla (AngularJS, 2015). Karman sivuilla on mainittu, että sillä voidaan ajaa ainakin Jasmine-, Mocha- ja QUnit-testejä. Karmaa voidaan myös käyttää jatkuvan integraation palvelimella, esimerkiksi Jenkinsin tai Travisin kanssa (Karma, 2015). Jatkuvalla integraatiolla tarkoitetaan menetelmää, jossa koko ohjelmisto kootaan ja integroidaan jatkuvasti.

7 Lopputulokset ja pohdinta

Opinnäytetyön alussa esitettiin tämän opinnäytetyön tavoite: selvittää, miten JavaScript-koodin yksikkötestaaminen tapahtuu ja että kuinka mielekästä JavaScript-koodin yksikkötestaaminen ylipäätään on. Tässä luvussa on käyty läpi, miten hyvin nämä tavoitteet saavutettiin ja pohdittu opinnäytetyöprosessia oman oppimisen näkökannasta.

Opinnäytetyön tekeminen osoitti, että JavaScript-koodin yksikkötestaaminen on haasteellista, mutta mahdollista. Kaikkien asioiden yksikkötestaaminen ei ole JavaScript-ympäristössä mielekästä, mutta pääosin ohjelmalogiikan testaaminen onnistuu. Käytännössä kokeillut yksikkötestauskirjastot, Jasmine ja QUnit, osoittautuivat molemmat toimiviksi. Käytännössä en havainnut näiden yksikkötestauskirjastojen välillä mitään paremmuuseroa.

Omaa oppimista opinnäytetyöprosessissa karttui merkittävästi. Kun tavoitteena oli tutkia yksikkötestaamista, ohjasi tämä myös itse prototyyppisovelluksen ohjelmointia: mietin jo sitä tehdessäni, kuinka yksikkötestattavaa koodini on. Käytännössä myös testausprosessin aikana tuli ilmi periaatteita, joilla ohjelmakoodista saa parempaa. Vaikka toteuttamani sovellus oli periaatteessa käyttökelpoinen jo ennen yksikkötestauksen aloittamista, voidaan sanoa, että ohjelmakoodin laatu parani yksikkötestausprosessin aikana. Opinnäytetyöni osoitti siis käytännössä, että yksikkötestauksella on koodin laatua parantava vaikutus.

Suurimpana haasteena opinnäytetyötä tehdessä oli ajankäyttö. Valitettavasti minulta ei liiennyt opinnäytetyölle niin paljon aikaa kuin olin alussa suunnitellut, ja sen vuoksi myöhästyin hieman alkuperäisestä tavoitteestani. Opinnäytetyön tekemisessä meni myös turhan paljon aikaa prototyyppisovelluksen laatimiseen, sillä tarkoituksena ei ollut tutkia JavaScript-ohjelmointia itsessään, vaan sen yksikkötestattavuutta. Myös itse yksikkötestien laatimiseen, testauskirjastojen opetteluun ja testien suunnitteluun meni loppujen lopuksi enemmän aikaa kuin olin suunnitellut. Uskoisin tästä kokemuksesta kuitenkin olevan jatkossa hyötyä esimerkiksi työelämässä.

Jatkotutkimusmahdollisuutena olisi tutustua aiheeseen hieman syvemmin. Tätä opinnäytetyötä tehdessäni sain hankittua pohjakokemuksen, jonka myötä JavaScriptin yksikkötestaukseen olisi helppo syventyä tarkemmin. Tällöin pääsisi myös tutustumaan tarkemmin esimerkiksi Node.js -ympäristöön (joka jäi lähinnä ajanpuutteen ja aikaisemman kokemuksen puutteen takia pois), testien automatisointiin ja vaikkapa

siihen, miten JavaScript-yksikkötestejä voidaan hyödyntää jatkuvan integraation yhteydessä.

Lähteet

AngularJS. Developer Guide / Unit Testing. Luettavissa: https://docs.angularjs.org/guide/unit-testing. Luettu 21.12.2015.

Flanagan, D. 2011. JavaScript: The Definitive Guide. O'Reilly Media.

Jasmine. Luettavissa: http://jasmine.github.io/2.3/introduction.html. Luettu 21.12.2015.

Jorgensen, P. 2013. Software Testing: A Craftsman's Approach, Fourth Edition. CRC Press.

Karma. Frequently Asked Questions. Luettavissa: http://karma-runner.github.io/0.12/intro/faq.html. Luettu 21.12.2015.

Mocha. Luettavissa https://mochajs.org/. Luettu 21.12.2015.

Pan, J. Software Testing. Luettavissa: http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/. Luettu 16.1.2016.

QUnit. Luettavissa: http://qunitjs.com/. Luettu 21.12.2015.

Saleh, H. 2013. JavaScript Unit Testing. Packt Publishing.

Sinon.JS. Luettavissa http://sinonjs.org/. Luettu 27.12.2015.

TutorialsPoint. Node.js – Introduction. Luettavissa: http://www.tutorialspoint.com/nodejs/nodejs_introduction.htm. Luettu 21.12.2015.

Ullman, L. 2012. Modern JavaScript – Develop and design. Peachpit Press.

W3Schools. Bootstrap Get Started. Luettavissa: http://www.w3schools.com/bootstrap/bootstrap_get_started.asp. Luettu 26.12.2015.

W3Schools. Bootstrap JS Modal. Luettavissa: http://www.w3schools.com/bootstrap/bootstrap_ref_js_modal.asp. Luettu 13.1.2016.

W3Schools. jQuery Introduction. Luettavissa:
http://www.w3schools.com/jquery/jquery_intro.asp. Luettu 21.12.2015.